MULTIPLE SERVICE PROVISION

This invention relates to a telecommunications service session control system, in particular one for implementing multi-party services of different types. It also relates to apparatus equipped with application programming interfaces and development tools for use in developing multi-party services.

The Internet, and in particular, the WorldWide Web ("the Web") has proved a successful system for distributing information and accessing applications remotely. All that is required for the user is a terminal equipped with a browser application program (such as Microsoft Internet Explorer (trade mark)) and a TCP/IP connection.

On the network side, servers store Web pages, in the form of Hypertext Markup Language (HTML) documents, which may be accessed from the client side using a uniform resource locator (URL). On receipt of a requested HTML document, the browser parses the HTML content and presents the document in a graphical user interface (GUI) on the user's terminal.

In addition to simple downloading of documents, browsers are able to transmit supplementary information, such as in an HTML form, to allow the customisation of information retrieved from a Web server. Common gateway interfaces (CGIs) provide access to server application programs which produce such customisation.

Notwithstanding the success of the Web, the services available lack interactivity. Interactive services are laborious to develop, and service support systems (billing, authentication, subscription, etc) need to be custom built for each service.

The Telecommunications Information Networking Architecture Consortium (TINA-C) is defining a model for the provision of telecommunications services. The model includes a service architecture layer and a network architecture layer, which are based on a distributed processing environment (DPE). The DPE is provided on distributed computational nodes connected via a transport network. The service architecture layer provides service control functions such as authentication, identification, subscription control, billing and service usage. The network architecture provides network connections to services and network management.

The TINA-C service architecture is itself split into two layers - the access layer and the service usage layer. The access layer is concerned with the provision of a secure communications link between a user and a notional service retailer, referred to as an "access session". From this access session, services can be accessed by the user in the context of a "service session" which includes a notional service provider.

In attempting to provide an implementation of TINA-C architecture, there are conflicting desiderata in relation to the provision for service development. On the one hand, the implementation preferably provides for a wide variety of possible services to be supported, thus placing fewer generic constraints in the system. On the other, services should be as simple and quick to implement as possible, thus favouring greater generic constraints in the system. Service developers may be aided by the use of a service session control system which takes away from the service developer the need to implement service generic functions.

The present invention is based on an object-oriented programming model. In object-oriented programming, software objects consist of information defining

properties of the objects which are both data variables and the methods (functions) which operate on those variables.

Objects are instances of a class, which is an object template insofar as it defines the variables and methods which the object possesses. An object is created by instantiating the class to which the object belongs.

Object classes may be arranged in a class hierarchy. A base class may be used to derive sub-classes by inheritance. The sub-classes inherit the methods and variables of the base class. Additional variables and methods may also be defined in the sub-classes. Sub-classes may also override, that is to say further define or modify, functionality defined in the base class. The sub-class defines a variable or method which has the same name as the property defined in the base class.

A set of base classes may be defined to provide a framework. A framework consists of a set of cooperating classes which makes up a reusable design for a specific type of software. A framework provides architectural guidance by partitioning the design into abstract framework classes and defining their responsibilities and collaborations. A developer customises the framework classes to a particular application by sub-classing and composing concrete classes.

Further explanation of a framework, and other object-oriented concepts may be found in "Design Patterns", Erich Gamma et al., Addison-Wesley, ISBN 0-201-63361-2: Elements of Reusable Object-Oriented Software, and the bibliography references given therein.

It would be desirable to provide a service session control system for implementing multiple interactive, multi-party services which are convenient to design and implement thereon, whilst providing the scope for the system to support a relatively wide variety of services. It would also be desirable to provide

an application programming interface, and a service development system for developing such services to be implemented on such a control system.

In accordance with one aspect of the invention there is provided a telecommunications service session control system comprising at least one server and in use interacting with software objects derived from an application programming interface, said application programming interface comprising:

a first framework object class for deriving service specific object classes to be instantiated on a client machine during participation in a service session;

a second framework object class for deriving service specific object classes to be instantiated on a server during a service session, said second class representing said service session; and

a third framework object class for deriving service specific object classes to be instantiated on a server during participation in a service session, said third class representing said participation.

In accordance with a further aspect of the invention there is provided a data store holding an application programming interface for use in developing multi-party services to be implemented on a telecommunications service session control system, said application programming interface comprising:

a first framework object class for deriving service specific object classes to be instantiated on a client machine during participation in a service session;

a second framework object class for deriving service specific object classes to be instantiated on a server during a service session, said second class representing said service session; and

a third framework object class for deriving service specific object classes to be instantiated on a server during participation in a service session, said third class representing said participation.

The invention provides an application programming interface which allows service developers to import functionality from the three defined framework classes, thereby easing service development. The nature of the three framework classes is such that the service specific coding required to implement a service is both flexible and meaningful to the service developer.

In accordance with a further aspect of the invention there is provided a service development system for generating service specific application parts to be implemented in a distributed manner in a telecommunications service session control system, said system comprising:

a service component constructor:

for storing data defining a plurality of framework components to be distributed between a client station and a server, and a plurality of customisation components for customising said framework components in a service-specific manner;

for generating data defining a first user interface for representing said framework components and said customisation components as icons on a visual display means; and

for defining relationships between said framework components and said customisation components to generate customised components by operations on said first user interface;

a control system simulator for simulating the interfaces and functionality provided by said telecommunications service session control system; and

a service tester:

for generating interactions between said control system simulator and said customised components; and

for generating data defining a second user interface for representing participation in a service session via said telecommunications service session control system, on a visual display means, in response to said interactions.

Embodiments of the present invention will now be described, by way of example only, with reference to the accompanying drawings wherein:

Figure 1 is a schematic block diagram of a communications system in accordance with an embodiment of the present invention;

Figure 2 is a schematic block diagram of system components in the client, retailer and service provider domains of the system illustrated in Figure 1;

Figure 3 is a schematic block diagram illustrating an embodiment of the arrangement illustrated in Figures 1 and 2, consisting of interrelated software objects;

Figure 4 is a diagram schematically illustrating a session lifecycle in accordance with an embodiment of the present invention;

Figures 5 to 13 illustrate interactions between the objects illustrated in Figure 3 during various procedures causing a change in the status of a user of the system;

Figure 14 is a diagram schematically illustrating interaction relationships between the service-specific components of Figure 3;

Figure 15 illustrates the inheritance relationships between service-specific object classes and framework object classes in accordance with an aspect of the invention;

Figure 16 illustrates an embodiment of service-specific software components providing a particular service supported by the system of the present invention;

Figure 17 is a schematic block diagram showing apparatus for developing services in accordance with an embodiment of the invention; and

Figures 18 to 20 are diagrams schematically illustrating graphical user interfaces provided by the apparatus of Figure 17.

Figure 1 is a block diagram illustrating a telecommunications system in accordance with an embodiment of the present invention. The system consists of a retailer server RS and a plurality of third party servers TPS1, TPS2 ... TPSn. User terminals T1, T2 ... Tn are connected to the retailer server RS via a data communications network, in this embodiment the Internet.

The third party servers TPS1, TPS2 ... TPSn are either remotely connected to the retailer server RS via communication links 2, which may be separate physical links and/or logical links, as shown, or are co-located with the retailer server RS.

Each third party server has access to a database TPDB1, TPDB2 ... TPDBn for the storage and retrieval of service-related data.

Although the servers RS TPS1, TPS2 ... TPSn are illustrated in Figure 1 as consisting of single servers, each may consist of one or more servers interconnected in a network. The servers each may be implemented on a computing resource, such as a workstation computer. Each of the user terminals T1, T2 ... Tn may be implemented in the form of a workstation computer, a net computer, a mobile communications terminal, etc.

Figure 2 schematically illustrates further details of the telecommunications system, exemplifying components present in a single user terminal, the retailer server and a single third party server. On the client side, a service browser application program 4 provides access and support to a range of services provided by the various third party servers TPS1, TPS2 ... TPSn via the retailer server RS. The retailer server RS includes an access control part 5, whereby a client terminal may initially access the system, an authentication part 6 whereby users of the system may be authenticated such as by means of a password checking procedure, and a subscription part 7 which maintains subscription data for users of the system.

During participation in a service the client terminal 5, the retailer server and the third party server include application program parts, including on the client side, a service-specific application program part 8, on the retailer side, a session control application program part 9 and on the service provider side, a service control application program part 10.

The retailer server, or server network, also includes a billing part 11 which handles the billing of event records by the session control application program part 9 during service sessions. A more detailed account of a possible billing mechanism is given in our copending International patent application, filed on even date herewith and entitled "Service Provision Support System" (agents reference J40367), the contents of which are incorporated herein by reference.

It is to be understood that the separation into retailer and service provider domains illustrated in Figure 2 is not necessarily a physical division, or a division of ownership or control between different servers. For example, the service provider

service control part 10 may be held on a server controlled by the retailer, such as that holding the retailer server RS itself.

Figure 3 illustrates an embodiment of the invention in which the system is implemented as a software object system. In this embodiment, each of the computing elements T1, T2 ... Tn; RS; TPS1, TPS2 ... TPSn illustrated in Figure 1 is supported by a distributed processing environment (DPE), whereby distributed software objects in different physical parts of the system may interact by the passing of messages, for example using CORBA (Common Object Request Broker Architecture) mechanisms, via data communications links.

In the following, reference will be made to the object-oriented language Java (trade mark), which is not intended to be limiting. Other, and indeed heterogeneous mixtures of object-oriented languages may also be used to implement the system.

The distributed object system includes service generic code and service specific code, which is distributed between the client side and the server side during a service session. The part of the system including service generic code is referred to herein as the service session control system.

Service specific code in the client domain is referred to herein as a "front" object. The front object in this embodiment is a Java applet which runs on the client machine. The front object 13 communicates with other system objects, to support and control a user's participation in a session. The front object 13 also provides service specific front end functionality, such as the service GUI and handling user input relating to the service.

Service specific code in the service provider domain is referred to herein as a "back" object. The back object 32 represents the session and its participants

and may interact with other back end systems, such as servers and databases in the service provider domain. It co-ordinates the interaction between participants in a session and stores service specific state information while a participant is suspended. The back object 32 can also exert control over a session, for example it can initiate suspend and exit for one or more participants. As will be described later, the back object 32 includes a number of cooperating software objects.

Service Session Control System - Client Domain Software Objects

A retail workspace object (RWS) 12 runs on the client machine and provides the client with a means of accessing services.

The RWS 12 may be a downloadable Java applet which runs in a known Java-enabled browser, or a stand-alone Java application program. The RWS allows the user to log on to the service session control system, to access service sessions by downloading front objects 13 and to log off via a GUI.

The RWS 12 is multi-threaded to allow front objects 13 to be used in parallel, both with each other and with the RWS 12. For example, a user may participate in two services simultaneously and at the same time receive, via the RWS 12, an invitation to join a third service. Thus, a user may have multiple front objects 13 of the same or different services running simultaneously on the client terminal.

When the user exits or suspends a session the relevant front object 13 is destroyed.

Service Session Control System - Retailer Domain Software Objects

A gateway object 14 provides an initial point of contact for user access to the system. It mediates the authentication of the user's identity and password

with the RWS 12. Thereafter, the RWS 12 connects to a user account object 18 and the gateway plays no further part.

The gateway 14 is multi-threaded to ensure availability. That is to say, the gateway 14 is able, during the processing of a logon request received from one client terminal, to process logon requests arriving from different client terminals.

A user administration object 16 provides functionality to create new user accounts.

The user account object 18 serves as a single user's account within the retailer domain. It holds a record of all the services to which the user is subscribed. The user account 18 also receives and stores, or obtains as needed, information from other objects in the retailer domain about any suspended sessions the user may have, all public sessions available and any session invitations the user may have received.

The user account object 18 also mediates interactions between the RWS 12 and the remainder of the retailer domain to start, join and resume sessions. It handles notifications from a session object 30 that sessions have been suspended or ended.

User account objects 18 are present in the retailer domain even when the user which they represent are logged off the system.

A user account manager object 20 creates and maintains the user account objects 18 and provides a reference to the appropriate user account object 18 when a user logs on to the system to start a communications session with the retailer server RS.

An authentication server object 22 uses stored user authentication data to authenticate log on requests.

A service provider portfolio object (SPP) 24 maintains information on available services, including the service provider identities and their network addresses, and which session manager 28 is to be used to support a service (this allows loads to be managed across a plurality of session managers 28).

A session manager object 28 prompts a session factory object 26, and a back factory object 34 (which is in the service provider domain) to instantiate session objects 30 and back objects 32 to support service sessions and then mediates between the user accounts 18, the session objects 30 and the back objects 32 to support further service interactions (e.g. invite, join, suspend, resume). There is one or more session manager objects 28 in the system.

The session object 30 is a service generic object (i.e. an object used in multiple different types of services) which controls a service session and co-ordinates the interaction of the objects involved in that session. It handles generic session behaviour such as invitations and suspensions. For example, when a client front 13 suspends a user's session, the session object 30 responds by calls on the user account 18 and the custom back 32 and by transmitting an event message to the event handler 31.

There is a single session object 30 instantiated per service session being controlled by the system. The session object 30 is created when the first participant starts the session and is destroyed when the last participant exits the session. The session object 30 is otherwise not destroyed, even if all its participants are suspended.

Service Session Control System - Service Provider Domain Software Objects

A back factory object 34 is provided in the service provider domain. Back objects 32 are instantiated by the back factory object 34 at the initiation of the session manager 28.

There is a single back object 32 per session object 30. Each custom back object 32 exists only as long as the session object 30 handling the same service session exists.

It should be mentioned that objects in the service session support system, in particular the gateway 14, the user account manager 20, the authentication server 22 and the service provider portfolio 24 may consist of multiple federated copies forming a single logical object, in order to provide scalability of the system.

Figure 4 illustrates an example of a session lifecycle. The session is begun when a first participant (participant1) starts the session. Throughout the lifetime of the session, various numbers of participants may join the session, suspend, and exit the session. The session exists until the last participant (in the example shown, participant3) exits the session, when the session terminates.

Figures 5 to 13 illustrate interactions between objects in the system during changes in the session-related status of a user of the system. Messages sent between objects are indicated by solid arrows, and are numbered to indicate the sequence in which the messages are sent. Event channels (indicated by dotted solid arrows) are used in various cases by the session object 30. Messages are sent by event channels to avoid blocking the session object 30, as would occur if request-response procedures were used, and which would impact on the service to other participants.

Several of the interactions between the RWS 12 and the user account 18 identify particular session instances. Passing direct references to the session objects 30 to the RWS 12 may compromise security. To avoid this, the sessions are identified by the RWS 12 using stored "cookie" references, previously passed to it by the user account 18, whereby the user account 18 is able to identify the actual session object 30 being referred to.

For clarity, the objects not involved in the interactions being described are omitted from Figures 5 to 13.

Referring to Figure 5, the gateway 14 is the initial point of contact for user access to the system. The RWS 12 initiates a data communications session with the gateway and sends a logon message with the user name and password supplied by the user. Provided these values are authenticated by the authentication server 22, the gateway 14 then retrieves a reference to the user's user account from the user account manager 20 and then contacts the appropriate user account 18 to confirm that the user is now logged on.

Further interaction then takes place between the RWS 12 and the user account 18 where the user account 18 returns user profile information including subscribed services, suspended sessions, public sessions and invitations received, access to which information the RWS 12 then makes available to the user via the client terminal.

Referring to Figure 6, when the user initiates a session via the client terminal, the RWS 12 sends a session starting message to the user account 18 specifying the service requested. In order to create a front object 13, the RWS 12 spawns a thread to download the appropriate front object, and downloads the appropriate front object 13 for the service from the retailer server RS.

The user account 18 obtains details for the service requested from the SPP 24, including the identity of the session manager 28 to support the service session, and sends a message to the session manager 28 to create a new session. The session manager 28 sends a message to the back factory 34 to instantiate a new back object 32 and sends a message to the session factory 26 to instantiate a new session object 30. The back factory 34 and the session factory 26 return references to the new back object 32 and session object 30 respectively, which are then used by the session manager 28 to instruct the new session object 30 to attach to the new back object 32. The session object's reference is also passed back to the front 13 subsequently via the RWS 12 to allow the front object 13 to attach to the session object 30.

The user account 18 then sends a join-service message to the session manager 28 which in turn sends a request-join message to the created session object 30 to cause the user to join a session.

The RWS 12 instructs the front object 13 to send an attach message to the session 30. This establishes the communications channel between the front object 13 and the session object 30. The session object 30 then sends a participant-joined message to the back object 32.

Referring to Figure 7, a user can join a session to which he has received an invitation. The RWS 12 transmits a joined-session message to the user account object 18, which passes a "cookie" reference to indicate which invite the user wishes to accept. The user account 18 obtains details of the service from the SPP 24 and sends a message to the session manager 28 to verify that the session is still current. The session manager 28 pings the session object 30 and the custom

back 32 to confirm this. Meanwhile, the RWS 12 spawns a thread to download the appropriate front object 13 for the selected service.

Once the session is verified by the session manager 28, the user account object 18 sends a join message to the session manager 28, which results in a request-join message sent by the session manager 28 to the appropriate session object 30. If the join request succeeds, the RWS 12 sends a message to instruct the front object 13 to attach to the session object 30, identified by the reference transmitted by the user account 18. The session object 30 then sends a participant-joined message to the back object 32, by event channel.

Referring to Figure 8, the suspension of a user from a session can be initiated by the front 13 sending a suspend message to the session object 30. The front object 13 then destroys itself. The session object 30 sends a participant-suspended message to the back object 32, by event channel, and marks the participant in question as suspended. The session object 30 also sends a message to the user account 18 to notify it of the user's new status.

Finally, the user account 18 sends a message to inform the RWS 12 of the new suspension, passing a cookie reference to the session 30, so that the user is able to resume the session thereafter.

Referring to Figure 9, an alternative session suspension procedure can be initiated from the custom back 32. In this case, suspension is initiated by the custom back 32 sending a suspend message to the session object 30. The session object 30 sends a suspend message to the front object 13 in question, which then destroys itself. The session object 30 proceeds by sending a participant-suspended message to the back object 32. The process continues as described in relation to Figure 8.

Referring to Figure 10, the RWS 12 is made aware of all suspended sessions for the user by the user account 18 immediately when the user is logged on, and throughout an access session. Therefore, when a user elects to resume a session, the RWS 12 sends a resume message to the user account 18, which indicates the session to be resumed, by means of a cookie reference. The user account 18 sends a verify-session message to the appropriate session manager 28, which pings the session object 30 and back object 32 to verify that the session is still in progress.

Meanwhile, the RWS 12 downloads the appropriate front object 13 for the selected service. Once the session manager 28 has verified that the session remains available, the RWS 12 instructs the front object 13 to send an attach-front message to the session object 30 to restore the front connection with the session object 30. The session object 30 then sends a participant-resumed message to the back object 32, by event channel.

By providing for suspension of participation in a service session as desired, in addition to complete existing from a service session, the state of the participation may be stored in the back object 32 for the duration of the suspension, and recovered when the participant opts to resume participation. Destruction of the front object on suspension allows for efficient management of the client terminal resources, as a new front object may be readily downloaded on resumption.

Referring to Figure 11, when a user wishes to exit a session, the front object 13 sends an end-participation message to the session object 30, and then destroys itself. The session object 30 sends a participant-left message to the back object 32. The session object 30 also transmits a message to the event handler

31 to inform it of the participant leaving event. The session object 30 then sends a message to the user account 18 to notify it that the user has left the session. The session object 30 then destroys itself if there are no active or suspended participants left in the session.

Exiting the session may alternatively be initiated, as a result of service logic processing, from the back object 32 as illustrated in Figure 12. In this case, exit is initiated by the back object 32 sending an end-participation message to the session object 30. The session object 30 sends an end-participation message to the front object 13, which then destroys itself. The session object 30 then sends a participant-left message to the back object 32. The process then continues as described in relation to Figure 11.

Referring to Figure 13, when the user logs off, the RWS 12 sends a logoff message to the user account 18.

An implementation of a service session may be considered in a service specific manner, as shown in Figure 14, without the generic components present. A service session consists of a single back object 32, deployed within the service provider domain, together with an arbitrary number of objects 13. One front object 13 is present per participation in the service. The front objects 13 may be distributed in several client terminals, and/or within a single client terminal . The interactions between the front objects 13 and the back object 32 are mediated by the remainder of the system.

Thus, a service developer need not be concerned when developing a service with implementing the multi-party session control functions provided by the system. All that the service provider needs to develop are the code for the service specific front object 13 and the code for the service specific back object 32.

In order to further simplify the task of the service developer, three framework object classes are provided, as part of an application programming interface (API) which the service developer uses by inheritance in their own service specific classes. The framework object classes are provided to a service developer, for example by the service developer downloading same from a networked server, in the form of traditional object classes, written in C++, Java or suchlike. The service provider may customise these classes, using a development tool, into front and back object classes which are service specific. The compiled classes may then be uploaded onto the retailer server RS for use as distributed objects in instances of the service.

Figure 15 illustrates the relationship between the actual classes which are to be instanced as the front object 13 and the back object 32, being Custom Applet 40, Custom Session 44 and Custom Participant 42, respectively, and the three framework object classes, being API Applet class 50, API Participant class 52 and API Session class 54.

Each of the three framework object classes include predefined methods which the service developer may call in their service code. Furthermore, in order to take advantage of the session control messages automatically generated by the service session control system, the service developer may override particular methods which are provided to receive automatically generated "callbacks" from the system in the framework classes but have no associated functionalities therein. When the service specific objects receive callbacks from the system, if the service developer has selected to override the callback methods therein, the service specific objects conduct the service specific functions coded by the service developer. Callback methods provided in the framework classes will be called by

the system, in particular the session object 30, as a result of specific occurrences during a session lifecycle.

Figure 16 illustrates the service specific software objects provided in a particular service session, exemplifying three separate participants. As can be seen, the back object 32 consists of a Custom Session object 60, and a number of Custom Participant objects 62, which have a one-to-one relationship with the Custom Applet objects 64 (the front objects) participating in the session.

The system, in particular the session object 30, automatically transmits callback messages to each of the fronts Custom Applet objects 64, each of the Custom Participant objects 62 and the Custom Session object 60 whenever a relevant session event occurs. For example, if a new participant joins, the back factory 34 is prompted to initiate a new Custom Participant object 62 to be included in the back object 32, as a result of the system procedures illustrated in Figure 6. Following the creation of the new Custom Participant object 62, and the corresponding Custom Applet object 64, the session object 30 generates *participantJoined* messages which are transmitted to each of the existing Custom Participant objects 62, and the Custom Session object 60.

Thus, generic session control messages, which are automatically generated by the system, may result in service-specific functions, coded by the service provider, being triggered in the service specific objects. The following provides a description of callbacks provided by the system and methods available on the framework classes.

Joining or Leaving a Session

justJoined and *justLeft* callbacks are provided on the API Participant class and the Applet class. These inform the derived object when the participant it represents has joined or left a session.

Inviting More Participants

A user starting a session or who has already joined a session may invite individual users. The method provided on the API Applet class is *invite*.

A user starting a session may invite all users logged on to the system. The method provided on the API Applet class is *publicAnnounce*.

Changes in Other Participant Status

Various callback methods are available both on the API Participant and API Session framework classes in order to enable the derived objects to determine changes in participant status. These callbacks are:

participantInvited

A participant already in a session has sent an invitation to a user, allowing them to join the session if they desire. A human readable description in the invitation may be included in the callback.

participantDeclined

A user has responded to an invitation by declining an offer to join the session. The invitation will have been sent out prior to this. A human readable description in the invitation may be included in the callback.

participantJoined

A participant has joined the session. This includes both people who join as a result of responding to an invitation or public announcement, and the first person into the session who joins it as a result of starting the session. The

callback includes the participant ID of the user. A human readable description in the invitation may be included in the callback.

participantLeft

A participant in the session has now left the session. They cannot rejoin without being re-invited. The callback includes the participant ID of the user.

participantSuspended

A participant has suspended their participation in the session, and may resume at a later date. The callback includes the participant ID of the user.

participantResumed

A participant has resumed their participation in the session, after suspending at some earlier time. The callback includes the participant ID of the user.

Locating Participants

The API Participant and API Session classes also provide a number of methods to help find participants in the session. These methods are:

totalParticipants

This method returns the total number of participants in the session.

resetParticipantIterator

This resets a participant iterator. Each instance of Custom Participant and Custom Session has its own separate iterator, and so resetting the iterator on one instance of Custom Participant, for example, does not have effect on the other instances.

nextParticipant

Get the next participant ID using the internal iterator.

participantById

Get further information about a participant given its participant ID (e.g. from a *participantJoined* callback).

getUserName

Get the username of a participant.

getParticipant

Get the participant ID of a participant.

Thus, given a participant ID from a callback, it is possible to obtain a

pointer to the corresponding Custom Participant, find out the participant's name, and call methods on the Custom Participant. An iterator method allows the object to identify and circulate round all the participants currently in the session. For example, from a particular instance of Custom Participant, it is possible to invoke a method on all other instances of Custom Participant. This is accomplished by using the iterator and checking the participant IDs to miss out the participant from which the code is invoked.

Messages

The framework object classes provide the ability to send messages, via the service session control system, from the back objects to the front objects, and vice versa. The following API methods are provided:

sendMessage

This is provided in each of the API Applet, API Session and API Participant classes. In the case of the method on the API Session class, a particular participant ID is specified.

messageReceived

This is callback method provided both on the API Session class and API Participant class, and is called when a message is received from a particular participant's front object.

getMessage

This method is provided on the API Applet class to create a thread awaiting a return message from the back object.

Messages may for example be sent from the back object to the front object, to inform the front whenever a participant joins, leaves, suspends, resumes, or is invited to the session. The participant status callbacks on the API

Participant class are overridden, and the *sendMessage* API method is used to send a corresponding message to the front object.

More complex parameters may be also sent using separators to put arguments into a string, and then using string tokenizers to separate the arguments at the front object.

The corresponding front object for this example may include a GUI which is provided with a label to display the message arriving from the back object. A thread is created in the front object to wait for messages using the *getMessage* method.

Using Personal Preferences

The API Participant framework class provides the ability to store information unique to a user of the system which can be retrieved, changed and stored each time the particular user participates in a particular service. The API calls to support this provide the storage and retrieval of a string, which may contain attributes or arguments. The two API calls are:

getServiceData

Retrieve service data for the participant.

putServiceData

Store service data for the participant.

These functions may be used in callbacks such as *justJoined* and *justLeft*. For example, to store a user's preferred alias it is possible to provide a string attribute in the Custom Participant class, and initialise the attribute from the service data in the *justJoined* method and store it again in the *justLeft* method, in case it had been changed during a participation in a session.

Ending and Suspending Participation

Participation in a session can be ended or suspended from both the back object or front object. The API methods common to the framework classes are as follows:

endParticipation

Ends the participation of a user in a session.

suspendParticipation

Suspends the participation of a user in a session.

The API Applet callback methods are as follows:

forcedEndParticipation

Called when a back object has called *endParticipation*.

forcedSuspendParticipation

Called when a back object has called *suspendParticipation*.

stop

This is the last called method on the Custom Applet object, allowing the code to shut down threads, etc.

The following examples relate to ending participation, but suspending is accomplished by similarly calling the corresponding API methods.

To end participants from the front object, the *endParticipant* call is used on the Custom Applet. This results in a callback to *justLeft* on the corresponding Custom Participant, as well as *participantLeft* on all other participants and the CustomSession. Finally *stop* is called on the Custom Applet, as a result of which it would be destroyed.

To end the participation from a Custom Participant, the *endParticipation* method is used. This will result in a *forcedEndParticipation* callback on the Custom

Applet, as well as a *participantLeft* callback on all other Custom Participants and the Custom Session. A similar call from Custom Session can also be used which takes the participant ID of the participant who is to leave the session.

Referring once again to the procedures illustrated in Figures 5 to 13, the procedures illustrated include single callbacks on the back object 32. As will be appreciated from the preceding description, those procedures described have been simplified, as the back object consists of a number of objects derived from the framework classes. The callbacks described will in fact be made on each of the objects in the back object, including the Custom Session object and each of the Custom Participant objects to allow the service developer to include code in each of the back object classes which implements functionality resulting from such callbacks. As will further be appreciated, the callbacks illustrated in Figures 5 to 13 are only a subset of the callbacks provided by the system, further such callbacks being described above.

The provision of two separate framework classes whereby to derive service specific classes to be used in a back object 32, the two distinct classes including a session object class and a participant object class, provide a flexible yet meaningful framework for service designers. The session object class may be used to control the session specific functions, and to maintain a session state for a service, whereas the participant object class may be used to control the service provided to individual participants in a session, to control participant-specific functions on the server side, to maintain participant-specific state information and to control the parameters of a service in accordance with individual participant preferences.

As the system provides a range of callbacks throughout the session lifecycle to each of the framework classes, the service developer is provided with the capability to distribute service logic amongst the front object and the back object as appropriate. This provides flexibility which is useful in supporting a wide variety of possible services. Some types of services will benefit from the concentration of service logic in the front object, whereas other types of service will benefit from the concentration of logic in the back object. For example, a simple messaging service which requires little storage of state information will more readily be implemented with the service logic concentrated primarily in the front object. On the other hand, a service which is more complex, which requires a higher degree of maintenance of state information and/or which may be subject to frequent suspensions by users, would tend to favour a concentration of service logic in the back object.

By providing the three separate API framework classes, and a suitable description of all the methods available thereon including the various callbacks, the service developer is relieved of the need to know details of the system on which the service code will run. For example, common service surround functions, such as authentication, billing and management functions, need no longer be considered during the service development phase.

The present invention also provides a visual, iconic, programming development tool for use by service developers to develop the service-specific application program parts to be implemented in the front and back objects of a service supported by the service session control system of the present invention.

Various component models may be used to implement the visual programming development software tool, including Microsoft's ActiveX, Opendoc

and Openstep (trademarks). The following embodiment of development tool uses the JavaBeans (trademark) component model, although it will be appreciated that other component models, and object-oriented programming languages, may be used to implement the development tool.

The development tool provided by the present invention allows the integrated development and testing of front and back application program parts. Referring to Figure 17, the development tool of this embodiment includes a service builder software application program 100, a tester software application program 200 and a control system simulator application program 300.

The development tool runs within the workstation or network of the software developer. In this embodiment, the development tool is installed in the working memory of a computer workstation 400 having known features, including a data processor, random access memory, read only memory, a hard drive memory, a display monitor, keyboard and a pointing device such as a mouse.

The development tool is JavaBeans-enabled. In this regard, the service development tool may be implemented in a similar manner to an existing JavaBeans-enabled design tool, such as Sunsoft's Java Studio, Borland JBuilder or Symantec's Visual Cafe 2 (all trademarks).

The service builder 100 allows the customisation of the framework JavaBeans to build the service-specific application program parts, as will be detailed below.

The control system simulator 300 contains components showing the interfaces and functionality provided by the service session control system for which the service-specific application parts are being developed. These components have the session lifecycle control, object factory, event message

handling and call back generation functions of the service-generic control system components described in relation to Figures 2 to 13.

The tester 200 provides a simulation of the client graphical user interfaces (GUIs) which would be presented in the client terminals connected to the control system during participation in the service being built, to allow the testing of services as they are built in the service builder 100.

Figures 18, 19 and 20 illustrate the GUI provided on the service developer workstation display in programming and testing modes of the development tool.

Figures 18 and 19 illustrate the GUI provided by the service builder 100. The GUI includes a JavaBeans palette 102, an applet composition window 104, a custom session composition window 106 and a custom participant composition window 108.

The palette 102 contains a standard range of customiser JavaBeans provided with the development tool. In addition, a service developer may add further JavaBeans to the palette.

The composition windows 104, 106, 108 contain, as a basic component, a framework applet JavaBean 110, a framework session JavaBean 112 and a framework participant JavaBean 114, respectively.

Each of the framework JavaBeans 110, 112, 114 is a mapping of the API framework classes previously described on to a JavaBean component. The API methods and callbacks provided on the API framework classes are mapped on to JavaBean properties and methods. The callback methods provided on the framework classes are mapped to unbound properties of the JavaBeans 110, 112, 114. Other JavaBeans dropped into the composition windows from the palette may be connected to these unbound properties to register for API event

notifications. The API methods of the framework object classes are mapped on to either bound properties which may be customised by the service developer using property tables provided for each of the framework JavaBeans 110, 112, 114, or, alternatively, methods which JavaBeans from the palette 102 dropped into the composition windows may call.

The framework JavaBeans 110, 112, 114 also have events and methods which are uncustomisable. There are those which represent the interactions of the framework classes with the service session control system as previously described. In the development tool, these uncustomisable events and methods and are called by components in the control system simulator 300.

As illustrated in Figure 19, as a service developer builds a service-specific application, JavaBeans from the palette 102 are dropped into the composition windows 104, 106, 108 and interconnected with the framework JavaBeans 110, 112, 114 to customise the framework JavaBeans. The service developer also sets properties of the framework JavaBeans 110, 112, 114 and properties of the other JavaBeans which are placed in the composition windows 104, 106, 108 from the palette 102.

As the service developer builds an application using the service builder 100, the service developer may test the service under development by use of the tester application program 200 in a testing mode of the development tool.

Figure 20 illustrates the GUI provided by the tester 200, which includes control buttons 202, 204, 206 which are used to start and close a simulated service session and to produce new potential participants which may be joined into the session and to resume previous participations. In this regard, button 202 is a

open start/close session button. Button 204 is a system logon button and button 206 is a resume participation button.

As a simulated service session is under way, operation of the buttons 202, 204, 206 cause the control system simulator 300 to instantiate new service specific objects from the customised framework JavaBeans visually programmed in the service builder 100, and to destroy them when desired. This results in the appearance and disappearance of client GUI windows 208 which illustrate the interactive display which would be provided on each of the terminals of users participating in the service session, were the service application parts being built actually implemented on the service session control system of the present invention.

The control system simulator 300 provides the underlying service-generic functionality of the service session control system of the present invention, whereby the service-specific application parts being built interact in the test mode of the developer tool. The customised JavaBeans application parts present in each of the composition windows of the service builder 100 are interpreted by the tester application program 200 in order to provide the client GUI windows 208.

The service developer may view and interact with the service displays which would be provided in the client terminals during an actual service session, thereby testing the application parts under development. To start a session, the service developer operates the start/close session button, which causes the tester 100, through the control system simulator 300, to instantiate an instance of each of the currently customised session framework JavaBeans, the currently customised participant framework JavaBean and the currently customised applet framework JavaBean. A single client GUI window 208 appears.

New participants may be joined into the simulated control system using the system logon button 204, causing additional client GUI windows 208 to appear in the tester GUI, which windows may be selected and viewed by the service developer. The service developer may simulate any of the actions a user of the service will during actual access and service sessions be able to perform.

For example, the service developer may simulate the invitation of a participant by interaction with one of the client GUI windows, which would result in invitation messages appearing in other of the client GUI windows 208. The service developer may then simulate acceptance of the invitation by interaction with one of the client GUI windows displaying the invitation.

The service developer may simulate suspension of a user's participation in the service, by interaction with one of the client GUI windows 208. Suspension results in disappearance of the client GUI window 208 from the tester GUI. Subsequently, the service developer may simulate resumption of participation by actuation of the resume participation button 206, which causes reappearance of the client GUI window 208 in the tester GUI.

Thus, the development tool provides the service developer with the ability to build a distributed application supported by the service session control system of the present invention, visually using iconic programming. In addition, due to the provision of the control system simulator application 300 and the use of the JavaBeans model, which allows interpretation instead of the need for compilation of the components being built, the service developer may test the service under development immediately and dynamically as the service is built. The effects of changes to the customisation of the framework JavaBeans 110, 112, 114 in the

service building mode are immediately apparent in the service testing mode, which the service developer may readily toggle to and from.

It is envisaged that various modifications and variations may be employed in relation to the above-described embodiment, and the scope of the invention is defined in the appended claims.